DS286 | 2016-09-16,21

L11-12: Hashmap

Yogesh Simmhan

simmhan@cds.iisc.ac.in

Slides courtesy Venkatesh Babu, CDS

Ideal Hashing

- Uses a 1D array (or table) table[0:b-1]
  - Each position of this array is a bucket
  - Capacity of the bucket is b
  - A bucket can normally hold only one dictionary pair: `<key, value>`
- Uses a hash function h that converts each key k into an index in the range [0, b-1].
  - h(k) is the "home bucket" for key k.
- Every dictionary pair is stored in its home bucket
  table[h(item.key)] = item

Ideal Hashing Example

- KVPs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is **table[0:7]**, b = 8.
- Hash function **h=key/11**
- Pairs are stored in table as below:

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- get, put, and remove take O(1) time.

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- Where does (99,k) go?
- Hash function causes us to go beyond table size
- **Simple fix**: do a “mod” with the bucket size by default
- h = (k / 11) % 8

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- Where does (26,g) go?
- Keys 22 and 26 have the same home bucket, are synonyms with respect to the hash function used.
- The home bucket for (26,g) is already occupied.

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- A **collision** occurs when the home bucket for a new pair is occupied by a pair with a different key.
- An **overflow** occurs when there is no space in the home bucket for a new pair.
  - E.g. each bucket in table can hold two values for same key, and more than 2 values for the key are inserted
- When a bucket can hold only one pair, collisions and overflows together.
- Need a method to handle overflows.

Hash Table Issues

- Choice of hash function.
- Overflow handling method.
- Size (number of buckets) of hash table.

Good Hash Function

- Quick to compute
- Distributes keys **uniformly** throughout the table
  - Each bucket has the same probability of the number of keys in the input range that will be hashed to it
  - E.g. h=k%b is a uniform hash function for keys in the range [0..r] … assuming all keys have equal probability of occurence
  - Buckets get ceil(r/b) or floor(r/b) items
- Difficult to find a good hash function

Hashing non-integer keys

- Find ways to convert the keys to integers
- Eg:
  - ASCII to int (add up chars)
    - Does not distinguish various permutations
    - listen/silent, rescue/secure, live/evil/vile/veil
  - Remove special chars (1020SERC1002)
  - Shift left and add: $h += c_i + c_{(i+1)} << 8$

Keys to Indices

- Hash function is combination of
  - Hash code map [key → integer]
  - Compression map [integer → [0, N-1]]

- A good hash function minimizes the probability of collisions

Popular Hash-Code Maps

- Integer cast
  - For numeric type 32 bits or less: directly interpret (e.g. after a mod)
  - Component sum: For type more than 32 bits (eg., long, double), add up the 32 bit components.
  - The above is not good for strings

Hash Code Maps

- Polynomial accumulation:
  - For strings of natural language, combine the char values (ASCII or Unicode), by viewing them as the coefficient of polynomial:
    - $a_0+a_1 x+ \ldots + a_{n-1} x^{n-1}$
  - The choice **x=33,37,39,41** gives at most 6 collisions on a vocabulary of 50K English words.
  - Polynomial is computed with Horners's rule at a fixed value of 'x'.

Horner's Rule

- Given the polynomial *p(x)*:

$$p(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n,$$

- Write *p(x)* as:

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x)\cdots)).$$

- Evaluate at $x = x_0$

$$\begin{aligned} p(x_0) &= a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-1} + b_n x_0)\cdots)) \\ &= a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(b_{n-1})\cdots)) \\ &\ \vdots \\ &= a_0 + x_0(b_1) \\ &= b_0. \end{aligned}$$

Why rewrite?

Compression Maps

- Use the remainder
  - $h(k) = k \bmod m$, k is the key
  - m the size of the table. Need to choose m
  - *E.g.* $m=b^e$ is bad
    - If m is the power of 2, $h(k)$ gives the e LSBs of k
    - All keys with the same suffix go to same bucket
- m prime (not too close to exact powers of 2) is good
  - Helps ensure uniform distribution
  - *or pick closest prime to fixed bucket size*

Example

- Hash table for n=2000 char strings
- Allowed average collisions = 3
- Choose m=701
  - A prime near 2000/3
  - And not near any power of 2

Open Addressing

- All elements are stored in the hash table
  - Elements to store <= capacity of table
- Each table entry contains either an element or *null*
- While searching for an element systematically **probe** table slots

Open Addressing

- Modify the hash function to take the probe number i as the second parameter
  - $h: U \times \{0,1,\ldots m-1\} \rightarrow \{0,1,\ldots m-1\}$
- Hash function, *h*, determines the sequence of slots examined for a given key
- Probe sequence for a given key *k* is :
  - $\langle h(k,0), h(k,1), \ldots h(k,m-1) \rangle$ - a permutation of $\langle 0,1,\ldots m-1 \rangle$

Linear Probing

- If the current location is occupied, try the next location

```
LPInsert(k)
  If (table is full) return error
  probe = h(k)
  while (table[probe] occupied)
    probe = (probe+1) mod m
  table[probe]=k
```

- Uses less memory than *chaining* (later in lecture)
- Slower than *chaining*: Elements tend to aggregate, hence insertion time increases proportionally.

Linear Probing – Example

- Home bucket h(k) = k mod 17
- Insert keys: 6, 12, 34, *29*, 28, *11*, *23*, *7*, *0*, 33, *30*, *45*

0				4				8				12				16
34	*0*					6	*23*	*7*			28	12	*29*	*11*	*30*	33

0				**4**				**8**				**12**				**16**
34	*0*	*45*				6	*23*	*7*			28	12	*29*	*11*	*30*	33

Lookup in Linear Probing

- Search for a key: Go to *(k mod 17)* and continue looking at successive locations till we find *k* or reach empty location.
  - Longer (unsuccessful) lookup time
  - Deletion?

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

Deletion

- Shift all elements to previous location?
  - Costly
- Instead, place marker at vacated location
  - neverUsed=false
- Lookup continues till neverUsed=true
- Insert puts element in first location with neverUsed=true, sets it to false
- Too many markers degrade performance → Rehash

Double Hashing & Random Probing

- Uses two hash functions: *h, p*
  - *h(k)* determines the position in table
  - *p(k)* determines the probe offset on unsuccessful search
- Test locations h(k), (h(k)+p(k))%b, (h(k)+2.p(k))%b, …, (h(k)+i.p(k))%b
  - *p(k)*=1 for linear probing
- May also use r(i) for i^{th} probe, which is **random probing** if r() is a *pseudo-random* generator
- Test locations h(k), (h(k)+r(1))%b, (h(k)+r(2))%b, …, (h(k)+ r(i))%b

Double Hashing

```
DoubleHashingInsert(k)
  if (table is full) error
    Probe=h(k); offset=p(k)
  while (table[probe] occupied)
    Probe=(probe+offset) mod m
  table[probe] =k
```

- If *m* is prime, we will eventually examine every position in table
- Distributes keys more uniformly than linear probing

Hashing with Chaining

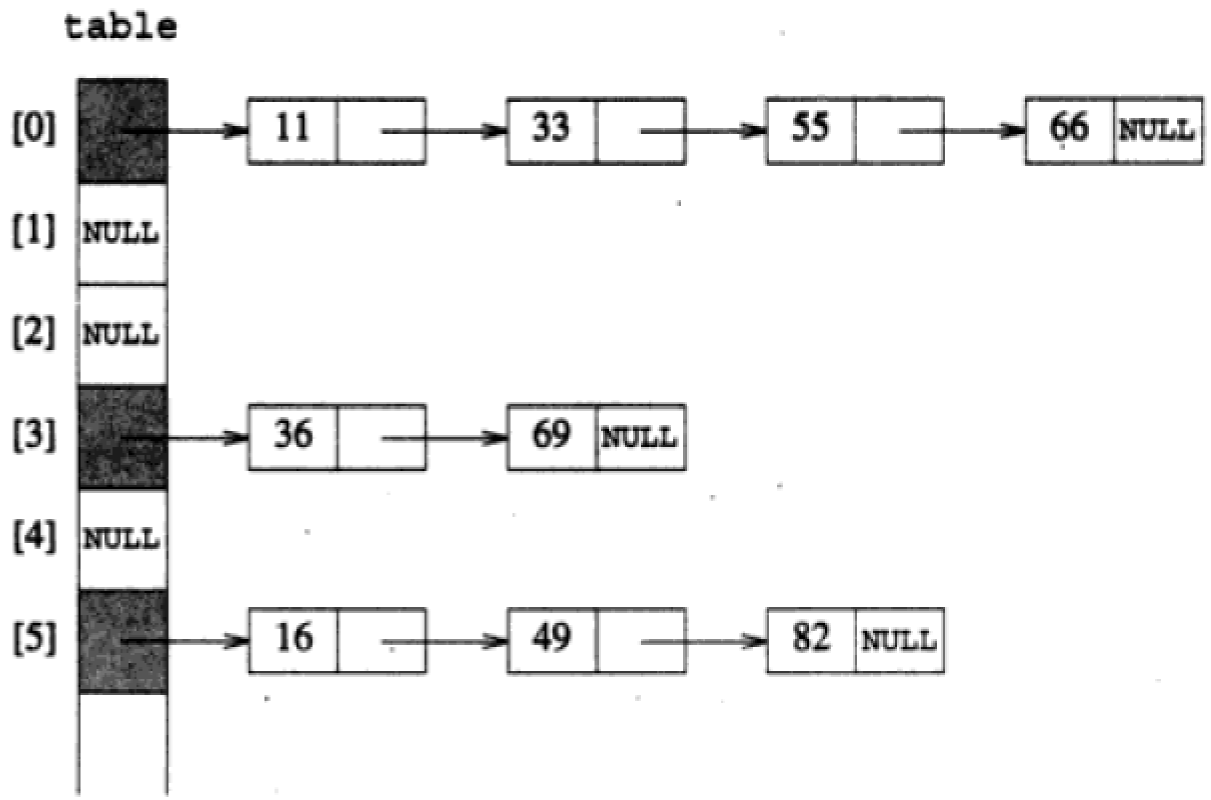

Figure 10.3 A chained hash table

Hashing with Chaining

- Collisions cause entry to be added to linked list
- O(1) insertion cost
- O(chain length) lookup, deletion cost
- More memory than array (pointers)
- Faster insertion

Analysis

- Load factor $\alpha = n/b$ is fraction of buckets occupied
- Assume that every probe looks at a random location in the table
  - linear probing/double hashing
- $1-\alpha$ fraction of the table is empty
- Expected number of probes to find an empty spot (unsuccessful search) is $1/(1-\alpha)$

Analysis

Expected number of un-successful trials given α

$$U_n \approx \frac{1}{p} = \frac{1}{1-\alpha}$$

Expected number of un-successful trials for i^{th} insert

$$\frac{1}{1-\frac{i-1}{b}}$$

Average number of trials for each of the *n* inserts

$$S_n \approx \frac{1}{n}\sum_{i=1}^{n}\frac{1}{1-\frac{i-1}{b}} = \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-\frac{i}{b}} \quad \ldots$$

$$= \frac{1}{\alpha}\log_e\frac{1}{1-\alpha}$$

Sahni, eqn. 10.5 and 10.6

Expected number of probes

	Unsuccessful	**Successful**
Chaining	$O(1+\alpha)$	$O(1+\alpha)$
Probing	$O(1/(1-\alpha))$	$O((1/\alpha)\log(1/(1-\alpha)))$

In chaining, α can be > 1

In probing, α is ≤ 1

Tasks

- Self study (Sahni Textbook)
  - **Check**: Have you read Chapter 10.1-10.4 “Dictionary and Skip Lists”? Solved problems?
  - **Read**: Chapter 10.5, Hashing from textbook
  - **Try:** Exercise 23, 26, 30 from Chapter 10 of textbook
- Finish Assignment 3 by Wed Sep 28 *(75 points)*

- 26 Sep (Mon) Class instead of tutorial
- 30 Sep (Fri) Institute holiday. But can we have class?
- Move Midterm from Oct 5 to Oct 7?
  - All lectures till Trees & Searching will be in syllabus

Questions?